netCDF Calculator User's Guide

Donald W. Denbo

Version 1.3
February 1999

Table of Contents

CHAPTER 1

Introduction

This manual describes a set of routines which provide data field manipulations with EPIC Data System data files. These routines form a library which can be linked with other programs, however, within the PPLUS scientific graphics system, they are available through the PPLUS *e* command. In addition to the interactive algebraic manipulations, the *e* command also provides interactive data editing for use with EPIC Data System data files. These files include EPIC observational data, such as time series and CTD profiles, as well as any data file written in one of the EPIC Data System supported formats, such as the multidimensional Unidata netCDF format. All original PPLUS commands and functions are unchanged and can be intermingled with the new *e* commands. PPLUS is an interactive scientific graphics program, and EPIC is an interactive system for management, display and analysis of oceanographic time series and hydrographic measurements. For more information about these systems, see the PPLUS and EPIC manuals. These EPIC System routines are available for use with PPLUS on VAX/VMS, Sun/Unix and DEC/Ultrix systems.

EPIC Data System files are data files in any of the formats supported by the EPIC Data System. These formats include the original EPIC formatted data files, which are binary sequential VAX/VMS files containing one dimensional oceanographic CTD data or time series data, and the newer EPIC implementation of the Unidata netCDF format, which allows data up to four dimensions (x,y,z,time) and is hardware independent under VAX/VMS, Ultrix and Unix (the same file can be read by hardware with any of these operating systems). The dimensions of (x,y,z,time) are

longitude, latitude, depth and time. The four dimensions can also be given by indices (i,j,k,l), which represent the same geographic and time coordinates. The latter notation is sometimes convenient for four dimensional model data. The geographic coordinates of (x,y,z,t) are more often useful for observational oceanographic data.

Algebraic manipulations available under the *e* command include arithmetic operations (addition, subtraction, multiplication and division), exponentiation, log functions, square root, and some additional functions such as differentiation, integration, regridding and removing the mean value. In addition to the built-in functions, the *e* command also allows user written routines for manipulation of data. The *e* command also provides the ability to extract a subset of the EPIC System Data sets by specifying limits in any or all of the four dimensions (x,y,z,time) or (i,j,k,l).

The *e* command includes an interactive data-editing function for one dimensional data sets. This displays the data and provides for replacement of individual points or a range of data points by linear interpolation or by typing in a replacement value. Data sets generated by the *e* data manipulation or data editing functions can be plotted with the standard PPLUS plotting commands and can also be written out as data files in any of the EPIC System formats.

Examples of algebraic manipulations and data editing of EPIC data files are included in this manual. Subsequent chapters describe the elements and syntax of the language of the PPLUS *e* command. It is recommended that the new user read the following Annotated Examples chapter, which includes a great deal of annotation and explanation. It assumes that the reader is familiar with both EPIC and PPLUS. The Annotated Examples chapter should allow the user to start using the *e* command. The later chapters describing the language itself may be more valuable as the user becomes more expert.

CHAPTER 2

Tutorial Overview

Introduction

This chapter provides a brief description of the netCDF Calculator (**nccalc**). It includes many examples for the user of the syntax and grammar of **nccalc**. There are more explicit and detailed descriptions of the language in the following sections on language elements and syntax. However, the new user may find that reading this chapter is enough to start using **nccalc**.

nccalc is a language in itself. Its data types include scalar, string, slab and field. A scalar is either a number, a variable representing a number, or a scalar array element. A string is a character string (deliminated by double quotes) or a character variable. A slab is a specification for a region of 4-dimensional space, for example, [x=95W,y=2S,z=50,t=*] is a slab representing the region with longitude 95W, latitude 2S, depth=50m and all times on the data file. A field is a multidimensional set of data, for example, temperature[x=95W:110W ,y=2S:2N ,z=0:50, t= 8601010000 : 8701010000] is a field of temperature values for the region 95W to 110W, 2S to 2N, depths between 0 and 50 meters, and times during 1986. The slab for this field can also be specified as [x=-95:-110,y=-2:2, z=0:50, t=198601010000 : 198701010000].

Specifying Time

nccalc is flexible in how time may be specified. Two basic formats are supported, the relative number of days since the start of the netCDF file and the Woods Hole date format. For example t=5.0, would indicated a time 5 days after the beginning of the file and t=198902010000, would indicate Feb 1, 1989 00:00 GMT. **nccalc** assumes that the first form if the value is less than 1000000.0 days. There are two legal forms for the WHOI date format, yymmddhhmm, where 1900+yy is the year, and yyyymmddhhmm, where yyyy is the year.

CHAPTER 3

Language Elements

Data Types

nccalc is a language in itself. Its data types include scalar, string, slab, field, and vector. This section of the manual gives a simple description of each of these data types. For a complete description of the language, its elements, and its syntax, see the appendix titled Reference Manual.

Scalar

A scalar is either a number, a variable representing a number, or a scalar array element. Scalars are distinct from fields in that scalar operators and functions operate on a single number at a time, whereas, fields have implied loops over all indices. Examples of scalars are the numbers 8, 12.4 and .007. In the statement:

```
a=3*12
```

the variable “a” is a scalar with value 36. The following example demonstrates the use of an scalar array, scalar arithmetic, and the scaler function “load”. In this example, “b” is defined as a scalar array, and the first two elements of “b” are set to values 1 and 2. The scalar variable “c” is set tothe product of these two scalar array elements, and then is loaded into the PPLUS symbol “eps$scaler” with the scalar function “load”.

```
array b[3]
b[1]=1
b[2]=2
c=b[1]*b[2]
load (c)

ppl>show eps$scaler
EPS$SCALER = 2
```

Scalar functions are functions which have scalars, scalar variables, or scalar array elements as arguments.

String

A string is a character string (deliminated by double quotes) or a character variable. The following is an example of a string: “This is a character string”. Here are two examples of string variables: a=”This is a character string” and b=”(a long string)”. Here is an example of string concatenation:

```
f=”This is a character string”
g=”(a long string)”
h=f+” “+g
```

The result of the string concactination is h=”This is a character string (a long string)”. The string can be used to set attributes in netCDF files and to construct file names for the **openr**, **openw**, and **nextw** commands.

Slab

A slab is a specification for a region of four-dimensional space. It has three geographical or geometrical dimensions and one time dimension. For example, [x=95W, y=2S, z=50, t=*] is a slab representing the region with longitude 95W, latitude 2S, depth=50m and all times on the data file. Another way to express a slab is [i=1, j=5, k=10, l=*], where the indices (i, j, k ,l) represent the coordinates (x, y, z, t). Another example of a slab is [x=95W:110W, y=10N:10S, z=200:300, t=8601010000 : 8701010000], where ranges of x, y, z, and t are indicated in the slab. The slab for this field can also be specified as [x=-95:-110, y=-2:2, z=0:50, t=198601010000 : 198701010000]. A slab variable is a variable to which a slab has been assigned, such as “a” in the following: a=[x=95W ,y=2S, z=50, t=*].

One slab may be modified by another slab. If slab a is defined as a=[x=95W ,y=2S, z=50, t=*], then another slab b can be created which is just like a, except that time is restricted to the year 1986, as follows: b=[a, t=198601010000 : 198701010000].

Field

A field is a multidimensional set, or subset, of data. The set or subset is defined by the slab. For example, “temperature[x=95W:110W, y=2S:2N, z=0:50, t=198601010000 : 198701010000]” is a field of temperature values for the region 95W to 110W, 2S to 2N, depths between 0 and 50 meters, and times during 1986. The field names (temperature in the preceeding example) are those available in the current database. Slab variables can be used to specify fields, as in this example, where sl is a slab variable and t1 is a field variable.

```
s1=[x=95W:110W,y=2S:2N,z=0:50,t=198601010000:198701010000]
t1=temperature[s1]
```

The following is an example of field algebra:

```
s1=[x=95W:110W,y=2S:2N,z=0:50,t=198601010000:198701010000]
u1=u[s1]
v1=v[s1]
spd=sqrt(u1*u1 + v1*v1)
```

Vector

```
e vector(uvel,wvel)
```

Operators

Binary operators are exponentiation (represented by ^), division (/), multiplication (*), addition (+), substraction (-), assignment (+), and logical operators of negation (!), greater than (>), greater or equal (>=), less than (<), less or equal (<=), logical AND (&&), logical OR (||), and not equal (!=).

Expressions

nccalc is an expression language, much like C. Expressions are scalar, string, slab, dbase, field, or file and variable attribute expressions. Expressions are the components of which statements are constructed. Examples of each type of expression are given here:

Scalar:

```
3
b
3*b
(3*b)
(3*b)/12
```

String:

```
“This is a character string”
a
```

Slab:

```
[x=95W,y=10N,z=300,t=198601010000:198701010000]
[x=95w:110W,y=-10,10,z=200:300,t=*]
[i=1:10,j=20,z=5,t=*]
[i=1:2*5,j=20,z=5,l=*]
s1
[s1, z=150]           !slab s1 modified to select z=150
```

dbase:

```
epsin                 !default for most current input file
epsin_2               !default for second input file opened
epsout                !default for current output file
epsout_3              !default for third output file opened
openr “/home/heron/taohome/data/t0n170w.cdf”!open for input
openw “output.file” “CDF”!open for output
```

field:

```
temp[z=*,t=*]
temp[i=1,j=1,z=*,t=*]
eps20[sl],[z=150]
t1                    t1 is a field variable
t1-t2                 !t1 and t2 are field variables
(t1-t2)/2.
sqrt( (t1-t2)/w) )    ! sqrt is a field function
```

file attributes:

```
epsin.CREATION_DATE
epsin.missing_value
epsout.missing_value
variable attributes:
wspeed.VARID
t1.VARID
t1.name
```

```
t2.long_name
```

Statements

Statements are composed of expressions. One common type of statement is the assignment statement. The following are examples of statements:

```
array b[3]            !declares b as an array of dimension 3
b[1]=1                !assigns value1 to second element of b
b[2]=2
c=b[1]*b[2]
load (c)              !function loads value of scalar c into
                      eps$scaler
f=”This is a character string”
g=”(a long character string)”
h=f+” “+g
s1=[x=95W:110W,y=2S:2N,z=0:50,t=198601010000:198701010000]
u1=u[s1]
v1=v[s1]
spd=sqrt(u1*u1 + v1*v1)
```

Control flow statements include while, if,and if-else. Examples follow:

```
if (x < 0) y=x else y=sqrt(x)
if (x < 0) {
 y = x
} else {
 y = sqrt(x)
}
```

Commands

In addition to statements, **nccalc** also includes several commands for manipulating symbols and performing file manipulations, such as opening, reading, and writing files. There are also commands for obtaining information about files and the data they contain. A complete list of commands appears in the Commands section of this document, but the following examples of some commands are given below:

```
openr “t2n170w.cdf”
nextr
status uspeed
whatis uspeed
close epsin
```

In the following example a new file is created and variable t1 is written to the new file. When creating a file the file must be explicitly closed with “close newfile” where newfile is the variable containing the file reference.

```
newfile = openw “pointer.out” “CDF”
nextw newfile “t2n170w.out”
write newfile t1
close newfile


load (4*3)
```

Logical structures

Logical structures consist of while loops, if and if-else statements. Examples follow:

```
while (xx < 20) {
b[1]=xx
xx=xx+1
}


if (xx < 0) yy=xx else yy=sqrt(xx)


if (xx < 0) {
yy = xx
} else {
yy = sqrt(xx)
}
```

Functions and Procedures

The only difference between functions and procedures is that functions return a value, and procedures do not.

Functions include log functions, square root, and some additional functions such as differentiation, integration, regridding and removing the mean value. There is also

an interactive data editing function, which allows a user to interactively edit the numerical values of a one-dimensional field. In addition to the built-in functions, nccalc also allows user written routines for manipulation of data. A complete list of all functions is in the section on Functions. Scalar functions return scalar values, and field functions return field values. Examples of some functions and their arguments are given here:

```
abs(xx)
atan(t1)
log10(a)
sqrt(b*3)
edit(salin)
fft(f)
regrid(f1,f2)
```

Built-in functions for fields

For more information about the built-in functions available for working with data fields, see the “Field” section of the “Expressions” chapter, which contains a table of these functions. They include the data editing function, integration,differentiation, calculation of a few oceanographic parameters and a regridding function.

CHAPTER 4

Functions

Table of Functions

TABLE 1. Functions

Function:	Returns:	Arguments:	Description:
abs(x)	scalar	scalar	returns the absolute value of x.
	field	field	returns a field containing the absolute value of each element in the original field.
atan(x)	scalar	scalar	returns the arc tangent of x
cos(x)	scalar	scalar	returns the cosine of x
exp(x)	scalar	scalar	returns e^x, exponential of x
	field	field	returns a field containing e^x of each element in the original field.
int(x)	scalar	integer	returns the integer portion of x, truncated towards zero
log(x)	scalar	scalar	returns ln x, logarithm base e of x
	field	field	returns a field containing ln x of each element in the original field

TABLE 1. Functions

Function:	Returns:	Arguments:	Description:
log10(x)	scalar	scalar	returns log x, logarithms base 10 of x
	field	field	returns a field containing log x for each element in the original field
sin(x)	scalar	scalar	returns sine of x
sqrt(x)	scalar	scalar	returns ??, the square root of x
	field	field	returns a field containing the square root of each element of the original field.

CHAPTER 5

Commands

APPENDIX A

Reference Manual

Expressions

nccalc is an expression language, much like C: although there are several control-flow statements, most statements such as assignments are expressions whose value is disregarded. For example, the assignment operator = assigns the value of its right operand to its left operand, and yields the value, so multiple assignments work. **nccalc** knows about four different data types: scalar (double precision floating point), string, slab (a specification for a four dimensional hyper-slab), and fields (a sub-sampled region of a four dimensional data set). The syntax for each of the four types are similar, however, not all operations are valid for all data types.

The **nccalc** command discussed here is a simple programmable interpreter for floating point, string and field expressions. It has been extensively modified from *hoc* as described in “The UNIX Programming Environment” by Kernighan and Pike.. It has C-style control flow, function definition and the usual numerical built-in functions. **nccalc** has been developed using lex, a lexical analyzer, and yacc, a parser generator. This allows a systematic and consistent syntax to be implemented easily.

Scalar

A scalar is a single or array of double precision floating point numbers. Scalars are distinct from fields in that scalar operators and functions operate on one number at a time, whereas, fields have implied loops over all indices. The expression grammar for scalars:

expr: *number*
| *coordinate*
| *variable*
| *arr-var* **[** *expr* **]**
| **(** *expr* **)**
| *expr binop expr*
| *unop expr*
| *function* **(** *arguments* **)**
| **scalar (** *fexpr* **)**

Numbers are floating point. The input format is that recognized by *scanf(3)*: digits, decimal point, digits, e or E, signed exponent. At least one digit or a decimal point must be present; the other components are optional.

Coordinates are floating point numbers that are followed by a single letter N, S, E or W. The sign of the number is changed if S or W are used.

Variable names are formed from a letter followed by a string of letters, underscores, and numbers. *arr-var* refers to a variable that has been declared to be an array. Arrays are only one dimensional and use 0-indexing. The syntax for creating an array is:

array *arr-var* **[** *expr* **]**

binop refers to binary operators such as addition or logical comparison; *unop* refers to the two negation operators, '!' (logical negation, 'not') and '-' (arithmetic negation, sign change).

TABLE 2. Scalar operators in decreasing order of precedence.

binop	description
^	exponentiation (FORTRAN **), right associative
! -	(unary) logical and arithmetic negation
/ *	division, multiplication
+ -	addition, subtraction

TABLE 2. Scalar operators in decreasing order of precedence.

binop	description
> >= < <= == !=	relational operators: greater, greater or equal, less, less or equal not equal (all same precedence)
&&	logical AND (both operands evaluated)
||	logical OR (both operands always evaluated)
=	assignment, right associative

Functions, as described later, may be defined by the user. Function arguments are expressions separated by commas. There are also a number of built-in scalar functions, all of which take a single argument, described below.

TABLE 3. Built-in scalar functions.

function	description
abs(x)	$\|x\|$, absolute value of x
atan(x)	arctan *x*, arc tangent of x
cos(x)	cos *x*, cosine of x
exp(x)	e^x, exponential of x
int(x)	integer part of x, truncated towards zero
log(x)	ln *x*, logarithm base e of x
log10(x)	log *x*, logarithm base 10 of x
sin(x)	sin*x*, sine of x
sqrt(x)	$\sqrt{x}$, square root of x

scalar is provided to convert field variable that contains only a single point to a scalar. If **scalar** is used with a non-single point field, the first point will be used.

Logical expressions have value 1.0 (true) and 0.0 (false). As in C, any non-zero value is taken to be true. As is always the case with floating point numbers, equality comparisons are inherently suspect.

TABLE 4. Built-in constants.

constant	value	description
DEG	57.295779513082	$180/\pi$, degrees per radian
E	2.7182818284590	*e*, base of natural logarithms

TABLE 4. Built-in constants.

constant	value	description
GAMMA	0.57721566490153	γ, Euler-Mascheroni constant
PHI	1.6180339887498	$(\sqrt{5}+1)/2$, the golden ration
PI	3.1415926535897	π, transcendental number
ERR_VERBOSE	1.0	print warning messages
ERR_FATAL	10.0	terminate on fatal condition
ERR_WARN	20.0	terminate on warning

String

Currently strings have a limited implementation. The expression grammar for strings is:

sexpr: *" string "*
| *str-var*
| *sexpr* **+** *sexpr*

A *string* is any series of characters that is enclosed by double quotes. The only legal operation is "+" which will concatenate two strings together. *Str-var* is a variable name to which a string has been assigned.

Slab

A slab is a specification for a region of four dimensional space. Three geometrical dimensions and one time dimension. Slabs are used to select the portion of the database that will be retrieved and stored in a field variable. The expression grammar for slabs is:

slexpr: *slab-var*
| *slxexpr*
| *slyexpr*
| *slzexpr*
| *sltexpr*
| *slexpr* **,** *slxexpr*
| *slexpr* **,** *slyexpr*
| *slexpr* **,** *slzexpr*
| *slexpr* **,** *sltexpr*

Slab-var is a variable name to which a slab has been assigned. *slxepxr, slyexpr*, etc... are single axis expressions defining the x, y, z and t extents, respectively. The syntax *slexpr , slxexpr* causes the slab defined by *slexpr* to have its x-axis range changed to that specified by *slxexpr*. The x-axis expression grammar is:

slxexpr: **x=** *expr* **:** *expr* **:** *expr*
| **x=** *expr* **:** *expr*
| **x=** *expr*
| **x= ***
| **x= * :** *expr*
| **i=** *expr* **:** *expr* **:** *expr*
| **i=** *expr* **:** *expr*
| **i=** *expr*
| **i= ***
| **i= * :** *expr*

The grammar for the y (j), z (k) and t (l) axes are similar. The first form of the grammar allows the specification of a specific range with a stride, the second a specific range, the third a single value, and the fourth the entire range of data available. The ranges can either be specified in terms of coordinates (x, y, z, and t) or index values (i, j, k, and l). The stride is an integral number of data points. No algebra is available for slabs.

Dbase

A dbase is the collection of epic system data files and the associated attributes. Multiple files may be read and written by using dbase expressions. The expression grammar for a dbase is

dbexpr: *db-var*
| **openr** *sexpr*
| **openw** *sexpr sexpr*

Db-var is a variable to which a dbase has been previously assigned. **openr** opens an epic system pointer file indicated by *sexpr* for reading. **openw** opens an epic pointer file indicated by the first *sexpr* for writing, the second *sexpr* indicates the type of file to open (e.g. “EPIC”, “CDF”).

The default dbase for reading is automatically assigned to the db-var **epsin** and for writing to **epsout**. The user can reassign these db-var’s in order to change the defaults. Each invocation of **openr** will also create db-var’s named **epsin_1**, **epsin_2**, etc. Each invocation of **openw** will create db-var’s named **epsout_1**, **epsout_2**, etc.

Field

A field is a multi-dimensional subset of data from a selected database. Operations performed on fields include all the data values in the field. The expression grammar for fields is:

fexpr: *field-name* **[** *slexpr* **]**
| *field-name* **[** *slexpr* **] (** *dbexpr* **)**
| *fld-var*
| *fld-var* **[** *slexpr* **]**
| *farr-var* **[** *expr* **]**
| **(** *fexpr* **)**
| *fexpr binop fexpr*
| *fexpr binop expr*
| *expr binop fexpr*
| *function* **(** *fexpr* **)**
| *function (fexpr* **,** *fexpr)*
| *function* **(** *fexpr* **,** *slab-var* **)**
| *function* **(** *fexpr* **, [** *slexpr* **])**
| *function* **(** *fexpr* **,** *dfopt* **)**
| **window (** *fexpr* **,** *slexpr* **,** *type* **,** *expr* **)**

Field-names are those available in the current database. For example, EPIC CTD files usually have temperature, salinity, dynhgt etc.. available. Field-names are either the generic variable name or the string “eps” followed by the variable code (e.g. eps20 is water temperature). If the optional *dbexpr* is not specified **epsin** will be used.

Fld-var is a variable to which a field has been previously assigned. *farr-var* refers to a variable that has been declared to be an array. Arrays are only one dimensional and use 0-indexing. The syntax for creating an array is:

field *farr-var* **[** *expr* **]**

It is also possible to declare a variable to be a field variable without specifying a slab. The field variable will be set to the shape of the field it is first associated with. The syntax for declaring a field variable is:

field *fld-var*

binop refers to binary operators such as addition.

TABLE 5. Field operators in decreasing order of precedence.

binop	description
^	exponentiation (FORTRAN **), right associative
/ *	division, multiplication
+ -	addition, subtraction
=	assignment, right associative

Exponentiation is only valid when raising a field to a scalar power. The other *binop*'s are valid for any combination of fields and scalars.

Field functions can not be defined by the user. There are built-in field functions, which take a one or two field arguments, described in Table 6 on page -23. In addition, there are built-in field functions, which take one field argument and a slab variable or [slab_expression] (which is used to define a range), described below.

TABLE 6. Built-in field functions.

function	description
abs(f)	$\lvert f \rvert$, absolute value of f
exp(f)	e^f, exponential of f
log(f)	ln *f*, logarithm base *e* of f
log10(f)	log *f*, logarithm base 10 of f
sqrt(f)	$\sqrt{f}$,square root of f
coast(f)	returns -1 if value > 10^{35} else 1
edit(f)	interactive editing of f (1-d only)
deriv(f)	derivative of f (1-d only)
intg(f)	integration of f (1-d only)
sum(f)	find the sum of all elements in the field
min(f)	find the minimum element in the field
max(f)	find the maximum element in the field
ave(f)	find the average of all elements in the field
monthm(f)	prints monthly means of f (1-d and TIME only)
demean(f)	remove the mean of f
spread(f1,f2)	spread the elements from f1 to fill the shape of f2

TABLE 6. Built-in field functions.

function	description
sigmat(temp,sal)	compute σ_t
theta(temp,sal)	compute Θ, potential temperature
sigma_theta(temp,sal)	compute σ_Θ, potential density
regrid(f1,f2)	regrid field f1 to the grid of f2

TABLE 7. Built-in field functions with range argument.

function	description
ave(f,range)	compute field averaged over range
prime(f,range)	compute perturbation field by averaging over range
max(f,range)	find the maximum element of each indicated range
min(f,range)	find the minimum element of each indicated range
sum(f,range)	find the sum of each indicated range
fft(f,range)	fourier transform of f over the indicated range
window(f,range,type,taper)	window f with window of given type and taper

max(f) and min(f) return the appropriate scalar. They also set variables to determine the position of the value returned. These variables are max_loc_x, max_loc_y, max_loc_z, and max_loc_t for maximums and the min equivalents for the minimum.

sum(f) returns a scalar.

The three functions with the slab expression return an array of an appropriate dimension. For example,

```
max(u,[x=*,t=*])
```

returns a y,z array of the maximum element in the x,t slab of each y and z.

spread(f1, f2) takes a field, f1, of lower dimension than f2, and spreads the values in f1 over the size of f2. For example, if f1 were an array of only one value, and f2 is a one dimensional array, the output is a 1 dimensional array the size of f2, and is filled entirely with the value in f1.

fft(f, range) returns the fourier transform of f in the dimensions indicated by range. For example, range = [x=*], will compute the fourier transform in the x-dimension and for range = [x=*,t=*], the two-dimensional fourier transform will be computed. The field is always demeaned using the range and optionally windowed if the "fft_window" and "fft_taper" string and scalar variables are set. The legal window types are **cosine, hanning, hamming**, or **bartlett. "**fft_taper" is only used for window type **cosine.** The string variable "fft_norm" is used to normalize the transfrom output. The legal normalization types are **spectra**, **density**, or **wavenumber**. **spectra**, the default, gives the raw fft output, the sum of the spectral values will equal the total series variance. **density** gives the spectral density of the fft, produced by dividing each component by the band width. The integral of the spectral density is the total series variance. Finally, **wavenumber** gives the spectral density multiplied by the wavenumber or frequency for each component. For example,

```
fft_norm="density"
fft_window="cosine"
fft_taper=0.1
fft(uvel, [x=*])
```

will compute the horizontal wave-number spectral density of uvel after applying a cosine window with a 10% taper.

window is provided to allow the user to externally apply a window to a field. The arguments are identical to those above.

TABLE 8. Built-in field functions with an option argument.

function	description
diffx(f,dfopt)	compute finite difference in x axis direction
diffy(f,dfopt)	compute finite difference in y axis direction
diffz(f,dfopt)	compute finite difference in z axis direction
difft(f,dfopt)	compute finite difference in t axis direction

The above finite difference operators diffx, diffy, diffz, and difft compute the differences in a single direction. More complicated difference operators can be built from

the kernel operations through recursive calls. The argument dfopt determines how the finite difference is calculated.

TABLE 9. dfopt values.

dfopt	difference calculation
CENTR1	$f_{i+1/2} = \frac{f_{i+1} - f_i}{\Delta_{i+1/2}}$
CENTR2	$f_i = \frac{f_{i+1} - f_{i-1}}{\Delta_{i+1/2} + \Delta_{i-1/2}}$

Where Δ is the grid separation in meters. If the grid is not in meters then Δ will be computed from the latitude and longitude of the coordinates of the grid points.

The binary field operators and the built-in field functions with two arguments will, depending on the value of the variable “error_opt”, give an error message and regrid the fields to a common grid if they are on different grids. The allowable values of “error_opt” are: 0, don’t stop or give an error message; “ERR_VERBOSE” don’t stop but do give an error message; “ERR_FATAL” only stop on fatal error; and “ERR_WARN” stop on warning.

Editing Fields

edit(*f*) allows a user to interactively edit the numerical values of a one-dimensional field. The dependent variable is point number. There are three **nccalc** variables that are used to control the display region of the edit function. They are “edit_min” and “edit_max” which are used to set the y-axis range and “edit_dpts” which is used to set the x-axis range.

There are eleven “buttons” to the right of the drawing area that when selected will allow the user to manipulate the data values and the display. The **NEXT**, **PREV**, **SKIP TO**, and **REDRAW** buttons allow the user to display the next section of data, the previous section, skip to a specific starting point or redisplay the current section, respectively.

A single point may be changed by first selecting a point by placing the cross-hairs over the data point and then either pressing the space bar if on a Textronix compatible device or a mouse button. The selected point will be indicated by drawing a

small circle around it. The user can then enter in a new value by selecting the **MISS** button (the data value will be set to 1.0E35), selecting the **CHNG VAL** button and then using the cross-hairs to indicate the new value (the new value will be graphically indicated with a small box), or selecting the **ENTR VAL** button and then entering the new value from the keyboard (the new value will be graphically indicated with a small box). The exact value of a data point can be determined by first selecting a point and then the **PRNT VAL** button.

A range of points may be changed at once by selecting the **SEL RNGE** button and then selecting a starting point and ending point with the cross-hairs (the selected points will be indicated by small circles). The selected range can then be set to the missing data value by selecting the **MISS** button or have a linear fit by selecting the **LINEAR** button.

To exit the function select the **QUIT** button.

File and Variable Attributes

The attributes that are stored as part of an EPIC data file are available to PPLUS via two symbols set by **nccalc**. These are “eps$attr_name”, the attribute name requested, and “eps$attr” the attribute value.

Attributes are also stored with the field variables. The variable attributes, *varid* (EPIC variable code), *name* (variable short name), *lname* (long name), *gname* (generic name), *units* (variable units) and *frmt* (variable suggested format) can be retrieved when a field variable or expression is used.

The expression grammar for attributes is:

atexpr: *fexpr* **.** *attr_name*
| *dbexpr* **.** *attr_name*
| **.** *attr_name*

Attr_names are those available in the current database or field variable.

The attributes associated with a field variable and a writable data file can be modified. Attributes can only be created for a writable data file.

The assignment grammar for attributes is:

atasgn: *fld-var* **.** *attr_name* **=** *sexpr*
| *db-var* **.** *attr_name* **=** *sexpr*

| *fld-var* **.** *attr_name* **=** *expr*
| *db-var* **.** *attr_name* **=** *expr*

When assigning *varid* to a field variable the value must be either a valid EPIC variable code or a number less than zero. If *varid* is less than zero the variable attribute information in the field variable will be used. NOTE: Use *varid* values less than zero only with netCDF file format.

Statements and Control Flow

nccalc statements have the following grammar:

stmt: *expr*
| *variable* **=** *expr*
| *sexpr*
| *str-var* **=** *sexpr*
| *slexpr*
| *slab-var* **=** *slab-var*
| *slab-var* **= [** *slexpr* **]**
| *dbexpr*
| *db-var* **=** *dbexpr*
| *fexpr*
| *fld-var* **=** *fexpr*
| *atexpr*
| *atasgn*
| *procedure* **(** *arglist* **)**
| **while (** *expr* **)** *stmt*
| **if (** *expr* **)** *stmt*
| **if (** *expr* **)** *stmt* **else** *stmt*
| **{** *stmtlist* **}**
| **print** *print-list*
| **return** *optional-expr*

stmtlist: *(nothing)*
| *stmtlist stmt*

print-list: *expr*
| *sexpr*
| *dbexpr*
| *print-list* **,** *expr*
| *print-list* **,** *sexpr*
| *print-list* **,** *dbexpr*

An assignment is parsed by default as a statement rather than an expression, so assignments typed interactively do not print their value.

Note that semicolons are not special to **nccalc**: statements are terminated by newlines. This causes some peculiar behavior. The following are legal **if** statements:

```
if (x < 0) y=x else y=sqrt(x)
```

```
if (x < 0) {
     y = x
} else {
     y = sqrt(x)
}
```

In the second example, the braces are mandatory: the newline after the if would terminate the statement and produce a syntax error were the brace omitted. **nccalc** does support multiple line commands, but the block of lines must start with the **begin** command and finish with the **end** command.

The syntax and semantics of **nccalc** control flow facilities are basically the same as in C. The **while** and **if** statements are just as in C, except there are no **break** or **continue** statements.

Commands

In addition to statements the **nccalc** system also has several commands for manipulating symbols and operating of eps files. The syntax for these are:

command:
- **clear** *var*
- **close**
- **close** *dbexpr*
- **nextr**
- **nextr** *dbexpr*
- **nextw** *sexpr*
- **nextw** *dbexpr sexpr*
- **write** *fexpr*
- **write (** *dbexpr* **)** *fexpr*
- **vector (** *fexpr* **,** *fexpr* **)**
- **line (** *fexpr* **,** *fexpr* **)**
- **load (** *expr* **)**
- **load (** *fexpr* **)**
- **load (** *fexpr* **,** *fexpr* **)**
- **load (** *ldexpr* **)**
- **status** *field-name*

| **status** *field-name* **(** *dbexpr* **)**
| **status** *fld-var*
| **whatis** *field-name*
| **whatis** *field-name* **(** *dbexpr* **)**
| **whatis** *fld-var*

The load expression grammar is:

ldexpr: **x=** *field-name*
| **x=** *fexpr*
| **y=** *field-name*
| **y=** *fexpr*
| **z=** *fexpr*
| **u=** *fexpr*
| **v=** *fexpr*

The memory storage of a variable of any type may be reclaimed by using the **clear** command. The **clear** command has a single argument that is a variable name. An eps pointer file and associated data file can be closed and storage reclaimed with the **close** command. If the optional *dbexpr* argument is not given **epsin** is used. An eps pointer file that has been opened for reading can be positioned to the next file with the **nextr** command. Again if the optional *dbexpr* argument is not given **epsin** is used.

An eps pointer file that has been opened for writing can have its current data file closed an the next assigned with the **nextw** command. The *sexpr* argument indicated the name of the data file to be opened for writing. If the optional *dbexpr* argument is not given **epsout** will be used.

A field described by *fexpr* can be written to the eps data file with the **write** command. If the optional *dbexpr* argument is not given **epsout** will be used.

The **whatis** command prints on the screen information about a field in a data file or variable. This command is interactive only and causes no information to be transferred to PPLUS.

The **status** command is similar to the whatis command except that information about a field in a data file or variable is loaded into PPLUS symbols. The following table lists those symbols.

The **load** command is used to load a scalar value into the PPLUS symbol “eps$scalar”. The symbol can then be used in PPLUS labels. The **load** command supersedes the functionality of the **line** and **vector** commands when the *load expression* gram-

mar form is used. In the load expression grammar, *x, y, and z* are used to determine what variable or axis will be used to for each axis, and *u and v* are used to determine vector field componets. This form of the **load** command will skip missing data for 1-d fields if the variable **load_clean** is non-zero.

The **vector** command is used to load two scalar fields into PPLUS as a single vector field. The first *fexpr* is the x component of the vector and the second *fexpr* is the y component. If the two fields are not on the same grid they will be regridded to a common grid.

The **line** command is used to load two scalar fields into PPLUS as a single line. The first *fexpr* provides the x values for the line and the second *fexpr* provides the y values for the line. If the two fields are not on the same grid they will be regridded to a common grid.

TABLE 10. PPLUS symbols set by nccalc when status is called.

symbol	description
eps$fld_name	Long name of variable
eps$fld_min	Minimum value of field (*fld-var* only)
eps$fld_max	Maximum value of field (*fld-var* only)
eps$fld_xmin	Minimum x axis value
eps$fld_xmax	Maximum x axis value
eps$fld_ymin	Minimum y axis value
eps$fld_ymax	Maximum y axis value
eps$fld_zmin	Minimum z axis value
eps$fld_zmax	Maximum z axis value
eps$fld_tmin	Minimum t axis value
eps$fld_tmax	Maximum t axis value
eps$fld_nx	Number of points in x axis
eps$fld_ny	Number of points in y axis
eps$fld_nz	Number of points in z axis
eps$fld_nt	Number of points in t axis

Input and Output

I/O - interaction with the keyboard

There is a requirement to be able to interact with the user, to have the calculator ask questions of the user when there is not enough information provided to fulfill the task. This will be done differently for the different interfacing programs using the calculator. Every change here will contain an ifdef loop dependant on whether the system is running through PPLUS, stand-alone, or with another program. We are also toying with using remote procedure calls to do this communication.

nccalc has the facilities to read in and write out both scalar and field data.

Scalar

The input function **read**, like the other built-ins, takes a single argument. Unlike the built-ins, though, the argument is not an expression: it is the name of a scalar variable. The next number (as defined above) is read from the standard input and assigned to the named variable. The return value of **read** is 1 (true) if a value was read, and 0 (false) if **read** encountered end of file or an error.

Output is generated with the **print** statement. The arguments to **print** are a comma-separated list of scalar expressions and strings, as in C. Newlines must be supplied; they are never provided automatically by **print**.

Note that **read** is a special built-in function, and therefore takes a single parenthesized argument, while **print** is a statement that takes a comma-separated, unparenthesized list:

```
while (read(x)) {
    print “value is “, x, “\n”
}
```

Field

An EPIC SYSTEM pointer file is assigned by the **openr** statement. The **openr** statement has a single quoted string as an argument. The string is the pointer file name. The pointer file is deassigned by the **close** statement. Fields are read from the database by using the *field-name [slexpr]* expression. Fields are loaded into PPLUS by using the *fexpr* statement. Field assignments do not load the field into

PPLUS. Currently, one and two dimensional fields can be loaded into PPLUS as lines and grids, respectively. Lines are plotted with the PPLUS “plot” command and grids with the “contour” command.

The following is an example of database specification and field input and output:

```
openr “epic.dat”
nextr
temp[z=*]             return a field to application
nextr
temp[t=198201010000:198212310000]returns a field
close

openr “tbs-8”
nextr
sl=[x=*,z=*,t=4.5]    get all x and z at 4.5 days
uvel=u[sl]
wvel=w[sl]
te=temp[sl]
wvel*te               returns heat flux
vector(uvel,wvel)     returns a two component vector
```

or

```
openr “tbs-8”
nextr
sl=[x=*,z=*,t=4.5]
w[sl]*temp[sl]        returns heat flux
vector(u[sl],w[sl])   returns a vector

openr “test-1.cdf”
nextr
sl=[x=*,z=*,t=5]
tp=prime(t[sl],[x=*])
wp=prime(w[sl],[x=*])
wptp=ave(wp*tp,[x=*])
wptp/scalar(w_star[t=5]*t_star[t=5])returns normalized heat
                      flux
```

The next EPIC SYSTEM data file in the pointer file list is accessed by the **next** statement. The following lists the PPLUS symbols set by *e*.

TABLE 11. PPLUS symbols set by nccalc when nextr is called.

symbol	description
eps$pointerfile	Pointer file name
eps$filetype	File format
eps$filename	Data file name
eps$datatype	Type of data (CTD, TIME)

TABLE 11. PPLUS symbols set by nccalc when nextr is called.

symbol	description
eps$varlist	List of variables in data file, separated with ;
eps$namelist	List of generic names in data file, separated with ;
ppl$eof	End of file read. YES if true, NO if false

.

TABLE 12. PPLUS symbols set when a field is loaded.

symbol	description
eps$field	Field name (short variable name)
eps$varname	Field name (long variable name)
eps$varunits	Field units
eps$latitude	Field latitude
eps$longitude	Field longitude
eps$depth	Field depth
eps$date	Date of field
eps$xtype	type of x axis, (LONE, LAT, DEPTH, TIME, TEMP, FIELD)
eps$xlabel	x axis label
eps$xunits	x axis units
eps$ytype	type of y axis, (LAT, DEPTH, SALINITY, FIELD)
eps$ylabel	y axis label
eps$yunits	y axis units
eps$label	field label
eps$lowercrnr	string consisting of the minimum x, y, and z
eps$uppercrnr	string consisting of the maximum x, y, and z
eps$xmin	minimum value of the x axis
eps$xmax	maximum value of the x axis
eps$ymin	minimum value of the y axis
eps$ymax	maximum value of the y axis
eps$tmin	minimum time (only valid if axis is time)
eps$tmax	maximum time (only valid if axis is time)

Functions and Procedures

Functions and procedures are distinct in **nccalc**, although they are defined by the same mechanism. This distinction is simply for run-time error checking: it is an error for a procedure to return a value, and for a function not to return one.

function: **func** *name***()** *stmt*

procedure: **proc** *name***()** *stmt*

name may be the name of any variable --- built-in functions are excluded. The definition, up to the opening brace or statement, must be on one line, as with the **if** statements above.

Unlike C, the body of a function or procedure may be any statement, not necessarily a compound (brace-enclosed) statement. Since semicolons have no meaning in **nccalc**, a null procedure body is formed by an empty pair of braces.

Functions and procedures may take scalar arguments, separated by commas, when invoked. Arguments are referred to as in the shell: $3 refers to the third (1-indexed) argument. They are passed by value and within the functions are semantically equivalent to scalar variables. It is an error to refer to an argument numbered greater than the number of arguments passed to the routine. The error checking is done dynamically, however, so a routine may have variable numbers of arguments if initial arguments affect the number of arguments to be referenced (as is C's *printf*).

Functions and procedures may be recursive, but the stack has limited depth (about a hundred calls). The following shows a **nccalc** definition of factorial:

```
func fac() if ($1<=0) return 1 else return $1*fac($1-1)
fac(10)
     3628800
fac(5)
     120
or

func fac() {
if ($1 <= 0) {
return 1
} else {
return $1 * fac($1 - 1)
}
}
fac(5)
     120
```

Reserved Keywords, Tokens, and Variables

nccalc reserves many keywords, tokens, and variables for its own use. These include function names, flow control keywords, built-in constants, etc.

TABLE 13. Keywords

Name	Description
array	Create an array of scalar values.
begin	Indicates the beginning of a multi-line command.
clear	Delete a variable from **nccalc**.
close	Close a netCDF file (read or write)
else	Second part of an **if - else** statement
end	Indicates the end of a multi-line command.
field	Create an array of field values.
func	Declares the beginning of a function definition.
if	The first part of an **if** or **if-else** statement
line	Superseded by load.
load	Load data into PPLUS buffers.
nextr	Read the next file.
nextw	Create a new file and add it to the pointer file list.
openr	Open a pointer/netCDF file for reading.
openw	Open a new pointer file for output.
print	Print to the screen.
proc	Declares the beginning of a procedure definition.
read	Read a scalar from the keyboard.
return	Return from a function.
scalar	Convert a field of length 1 to a scalar.
status	Loads information about a field or variable into PPLUS.
vector	Superseded by load.
whatis	Prints on screen information about a field or variable.
while	The first part of a **while** block.
window	Applies a window to a field, used with spectral routines.
write	Write fields and meta data to the specified with **nextw**.

TABLE 14. Tokens

Token	Description
x=	X or latitude of a range or x axis variable.
y=	Y or longitude of a range or y axis variable.
z=	Z or depth of a range.
t=	Time component of a range.
i=	X index of a range.
j=	Y index of a range.
k=	Z index of a range.
l=	T index of a range.
u=	Define u component of a vector.
v=	Define v component of a vector.

TABLE 15. Variables

Name	Description
DEG	Degrees per radian
DefaultSlab	The default slab to use for fields, can be overridden.
E	Base of natural logarithm.
ERR_FATAL	Terminate on fatal condition.
ERR_VERBOSE	Print warning messages.
ERR_WARN	Terminate on warning.
GAMMA	Euler-Mascheroni constant.
PHI	The golden ratio.
PI	Transcendental number.
edit_dpts	Set x-axis range for the **edit** function.
edit_max	Set y-axis maximum for the **edit** function.
edit_min	Set y-axis minimum for the **edit** function.
epsin	Current file open for reading.
epsout	Current file open for writing.
error_opt	Error option for file reading.
fft_norm	fft normalization option.
fft_option	fft demeaning option (e.g. “nodemean”).

TABLE 15. Variables

Name	Description
fft_taper	fft window taper.
fft_window	fft windowing function
load_clean	If non-zero do not load missing data to PPLUS.
max_loc_t	Time location of maximum field value (**max**).
max_loc_x	X location of maximum field value (**max**).
max_loc_y	Y location of maximum field value (**max**).
max_loc_z	Z location of maximum field value (**max**).
min_loc_t	Time location of minimum field value (**min**).
min_loc_x	X location of minimum field value (**min**).
min_loc_y	Y location of minimum field value (**min**).
min_loc_z	Z location of minimum field value (**min**).

TABLE 16. Functions

Name	Description
abs	Absolute value
atan	Arc tangent
ave	Average
coast	Returns -1 if value > 10^{35} else returns 1
cos	Cosine
demean	Remove the mean
deriv	Derivative
difft	T finite difference
diffx	X finite difference
diffy	Y finite difference
diffz	Z finite difference
edit	Interactive edit of 1-d field
exp	exponential
fft	Fourier transform
int	Integer part
intg	Integration

TABLE 16. Functions

Name	Description
log	Natural logarithm
log10	Common lograithm
max	Maximum value
min	Minimum value
monthm	Prints monthly means of field
prime	Perturbation field
regrid	Regrid field
sigma_theta	Potential density
sigmat	Sigma - T
sin	Sine
spread	Spread elements of field to fill shape
sqrt	Square root
sum	Compute sum
theta	Potential Temperature

APPENDIX B

Annotated Examples

EPIC

Read and Plot Data

The following is a simple example of a PPLUS command file which reads a temperature field from an EPIC formatted time series file. The EPIC pointer file, which is a list of data file names, is named T95W.DAT. The generic EPIC variable code for temperature is “temp”. The EPIC variable temperature could have been indicated equally well by the numeric EPIC key code of 20; to do this, replace “temp’“by “eps20” in the following PPLUS command file. The EPIC key file used by the *e* command is the file ep_key:epic.key. You can type or print this file, or look at it with the VAX editor (in read mode).

```
e openr “t95w.dat”    Opens EPIC pointer file for read
e nextr               Points to the first data file in the
                      list
e temp[t=198601010000:198701010000]Times are yyyymmddhhmm.
                      Read temperature field for 1986
plot                  Plot the temperatures
```

Calculations with Data

In this section there are examples of calculations with data fields. The concepts used are e fields, slabs, variables and operators. An example of a field is all the temperatures measured by a mooring on a current meter, or all of the temperatures measured by a CTD.

Calculation of wind speed

In the next example, zonal and meridional components of wind are read from an input time series data file, and wind speed is calculated and plotted. Zonal wind with a generic variable code of “u” is read into a variable named “uw’“and meridional wind, with a generic variable code of “v” is read into the *e* variable named “vw”. The “[t=*]” is a “slab”, or a specification for a region of four dimensional space. This example reads data from a one dimensional EPIC formatted file which is time series of temperature, and this slab expression specifies all times on the input data file. For a four dimensional netCDF EPIC data file, it could have included specifications for x (longitude), y (latitude), and z (depth), e.g., [y=-5:5, x=170:180, z=0:500, t=198701010000 : 198801010000]. The variable wnd_spd is the calculated wind speed. In this example, the *e* commands are intermingled with the standard PPLUS commands taxis, ylab, and plot.

```
e openr “wind_95w.dat” Opens EPIC pointer file for read
e nextr               Points to the first data file in the
                      list
if eps$data_type .eq. ‘TIME’ then
taxis,on              Turn on time axis before
else                  loading data for plotting
taxis,off
endif
e uw=u[t=*]           Read u values for all times
e vw=v[t=*]           Read v values for all times
e wnd_spd=sqrt(uw^2+vw^2)Calculate wind speed
e wnd_spd             Load wind speed into PPLUS plot buffer
ylab,wind speed, m/s  Label y-axis
plot                  Plot the wind speed
```

Notice that the PPLUS global symbol eps$data_type has been set by the “e nextr” command. See the Chapter “Input and Output” section on “Fields” for PPLUS global symbols set by *e* commands.

Calculate average over several data fields

In following example, a mean salinity profile is computed from the salinity fields on all the CTD data files listed in the input EPIC pointer file. Depths are restricted

to the range 0 and 500 db. Then plot the mean value and overplot the individual salinities. Notice that many PPLUS commands are intermingled with *e* commands. Use is made of the PPLUS symbol “c” and of the *e* command variable “sum”.

```
e openr “ctd95w.dat”  Opens EPIC pointer file for read
set c 0               Initialize PPLUS symbol for use as
                      counter
e sum=0               Initialize e variable for accumulating
                      sum
c Start while loop
while ppl$eof .ne. “YES” thenLoop over all files
e nextr               Points to the next data file in the list
inc c                 Increment the counter
e s’c’=salt[z=0:500]  Read in salinities for 0 to 500 meters
e sum=sum+s’c’        Accumulate the sum of the salinities
e s’c’                Load c-th salinity into PPLUS plot buff-
                      ers
endw                  End of while loop
c End while loop
e avg=sum/(c-1)       Calculate the average
e avg                 Load average into PPLUS plot buffers
line,’ppl$line_count’,,0Make this a solid line
yaxis,500,0,100       Set y-axis limits
xlab,Salinity         Label x-axis
plot,Mean Salinity    Plot mean and individual salinities
```

Time Series

The following will open the EPIC pointer file t95w.dat and compute differences between the temperatures.

```
ppl>e openr “t95w.dat”
ppl>e nextr
ppl>e sl=[t=198201010000:198312310000]
ppl>e t1=eps20[sl]
ppl>e nextr
ppl>e t2=eps20[sl]
ppl>e t1-t2
ppl>plot

ppl>e openr “t95w.dat”
ppl>e nextr
ppl>e openw “out_edit.dat” “EPIC”
ppl>e nextw “out_file.001”
ppl>e temp=edit(eps20[t=*])
ppl>e write temp
ppl>e close epsout

ppl>e t95w = openr “t95w.dat”
ppl>e t110w = openr “t110w.dat”
ppl>e nextr t95w
ppl>e nextr t110w
```

```
ppl>e sl=[t=198201010000:198312310000]
ppl>e eps20[sl](t95w) - eps20[sl](t110w)
ppl>plot
```

CTD

Data Editing

The *e* command includes an interactive data editing function, which allows a user to interactively edit the numerical values of a one-dimensional field. It is initiated with a PPLUS command like the following: “e temp_edit = edit(eps20[z=*])”. This command edits the EPIC variable temperature (eps20 means epic key code 20, which is the EPIC key code for temperature). It puts the edited temperature field values into the *e* variable “temp_edit”. Once the edit function is invoked, the user is put into graphics mode. The x-axis is point number and the y-axis is the variable being edited. Editing selections appear on the right hand portion of the screen.

The editing selections are in the form of eleven “buttons” to the right of the drawing area. These can be selected by the user to manipulate the data values and the display. The **NEXT**, **PREV**, and **REDRAW** buttons allow the user to display the next section of data, the previous section or redraw the current section. The **SKIP TO** button lets the user skip to a desired point number.

A single point may be changed by first selecting a point. Do this by placing the cross-hairs over the data point, and then either pressing the space bar (if on a Tektronix compatible device) or a mouse button (on an X device). The selected point will be circled to indicate it has been selected. The user can then: (1) enter in a new value by selecting the **MISS** button (the data value will be set to 1.0E35), (2) select the **CHNG VAL** button and use the cross-hairs to indicate the new value (a small box will be drawn around the new value), (3) select the **ENTR VAL** button and then enter the new value from the keyboard (a small box will be drawn around the new value). The exact value of a data point can be determined by first selecting a point and then the **PRNT VAL** button.

A range of points may be changed at once by selecting the **SEL RNGE** button and then selecting a starting point and ending point with the cross-hairs (the selected points will be circled). The selected range can then be set to the missing data value by selecting the **MISS** button, or be replaced by a linear fit between the good adja-

cent points by selecting the **LINEAR** button. The **REDRAW** button always redraws the display.

NOTE: Notice that to work with a single data point, first select the point, then select the buttons to define the action to be taken. In contrast, to work with a range of data points, first select the button **SEL RNGE**, then select the first and last data point in the range, and finally, select the buttons to define the action to be taken.

To exit the edit function, select the **QUIT** button.

CTD data editing

The example in this section illustrates the data editing function in the *e* command. Here, temperature from a CTD data file is edited with the *e* editing function. The input pointer file is opened and then the pointer is positioned at the first data file, which contains the temperature to be edited. An output pointer file and data file are opened for write to receive the edited temperature. Once the temperature is edited, it is written out to the new data file. The EPIC variables in the original CTD data file are listed with the PPLUS command SHOW EPS$VARLIST. They will be identified in the list by their EPIC variable key codes. A single PPLUS *e* statement both reads and writes out each of these variables.

```
e openr “cast8.dat”   Open EPIC pointer file for read
e nextr               Position pointer on next (first) data
                      file
e openw “final8.dat” “EPIC”Open pointer file for write
e nextw “tw287c008.edt”Name next data file to be written
e temp=edit(eps20[z=*])Edit temperature field from input data
                      file
e write temp          Write out edited temperature
show eps$varlist      List EPIC key codes for variables in
                      data file
e write eps41[z=*]    Write out variable from input
e write eps70[z=*]    Write out variable from input
e write eps60[z=*]    Write out variable from input
e write eps10[z=*]    Write out variable from input
e write eps50[z=*]    Write out variable from input
e write eps30[z=*]    Write out variable from input
e write eps71[z=*]    Write out variable from input
e write eps110[z=*]   Write out variable from input
e write eps111[z=*]   Write out variable from input
e write eps112[z=*]   Write out variable from input
e close epsout        Close output pointer file
e close epsin         Close input pointer file
```

CTD data editing with an *e* variable for the slab definition

The next example simplifies the preceding example by assigning a variable "s" the value of the slab [z=*], and illustrates the use of an *e* variable for a slab definition which will be used repeatedly.

```
e openr "cast8.dat"   Open EPIC pointer file for read
e nextr               Position pointer on next (first) data
                      file
e openw "final8.dat" "EPIC"Open pointer file for write
e nextw "tw287c008.edt"Name the next data file to be written
e s=[z=*]
e temp=edit(eps20[s]) Edit temperature field from input
e write temp          Write out edited temperature
show eps$varlist      List EPIC key codes for variables in
                      data file
e write eps41[s]      Write out variable from input
e write eps70[s]      Write out variable from input
e write eps60[s]      Write out variable from input
e write eps10[s]      Write out variable from input
e write eps50[s]      Write out variable from input
e write eps30[s]      Write out variable from input
e write eps71[s]      Write out variable from input
e write eps110[s]     Write out variable from input
e write eps111[s]     Write out variable from input
e write eps112[s]     Write out variable from input
e close epsout        Close output pointer file
e close epsin         Close input pointer file
```

Multi-dimensional Data

PPL$EXAMPLES:CDF.DAT is a pointer file for a Unidata netCDF EPIC System data file (TBS-8.CDF) containing data from a 2-d deep convection model. (This means that the ascii file PPL$EXAMPLES:CDF.DAT contains a single line with the data file name TBS-8.CDF.) EPIC generic variable field names are found with the EPIC EPS variable key codes in the file EP_KEY:EPIC.KEY. The following describes the data in this data file:

TABLE 17.

Index	Variable	Description	Limits
i	x=distance	(meters)	30m to 7650 m
j			
k	z=depth	(meters)	30m to 1700m
l	t=time	(days)	3.8 days to 5 days (hourly)

TABLE 18.

Variable Fields	EPS Code	Generic Name
zonal velocity	eps326	u
vertical velocity	eps328	w
temperature	eps20	temp
salinity	eps40	sal

Extracting a plane of data

The following PPLUS commands illustrate the use of the *e* command to slice 3-dimensional data set and produce a labeled contour plot of temperature with time on the xaxis, depth on the y-axis. Notice that the slab definition is made with all times and all depths indicated by t and z, whereas the index i is used to indicate that only the first distance is to be extracted. In this slab definition, the time and depth selection could have been k=* and l=* instead of t=* and z=*.

```
e openr “cdf.dat”     open the pointer file
e nextr               point to the first data file
e (alldep=temp[t=*,z=*,i=1])extract desired slice of data
xlab,’eps$xtype’      use symbol to label the x-axis
ylab,’eps$ytype’      use symbol to label the y-axis
contour ‘eps$label’   use symbol to label the plot
```

Extracting a line of data

The following code extracts a time series of u and a time series of v, calculates speed (square root of u squared plus v squared) and plots it. It also illustrates the use of an *e* variable assignment for the slab value of [i=1:1,t=*,k=2:2]. Note that the slab definition can contain mixed indices and space-time axis references.

```
e openr “cdf.dat”     open the pointer file
e nextr               point to the first data file
e s=[i=1:1,t=*,k=2:2] assign the slab a variable name s
e u1=u[s]             extract desired time series
e v1=v[s]             extract desired time series
e speed=sqrt(u1^2+v1^2)calculate speed from u and v
e speed               load speed into PPLUS plot buffer
taxis,on              turn on the time axis
ylab,Calculated Speed label the y-axis
plot Speed at i=1, k=2plot the calculated speed
```

Reading and Writing EPIC files

This section describes methods of reading information, such as header information or attributes from EPIC data files. It also describes techniques for writing attributes and data into EPIC data files. In addition, it describes exchanging the information between *e* function variables and PPLUS symbols. Examples will be used to illustrate these techniques. See the Expressions Chapter, which has a section on File and Variable Attributes, for more general information.

To retrieve the value of an attribute from an EPIC system data file, use the *e* function to open the file, and proceed as illustrated below:

```
ppl>e openr "t95w.dat"Open the file
ppl>e nextr           Read attributes
ppl>e epsin.DATA_ORIGINPut value of this attribute into PPLUS
                      symbol eps$attr
ppl>show eps$attr
```

In the preceding example, t95w.dat is an EPIC pointer file, containing a list of EPIC formatted time series data files. The value of the attribute named DATA_ORIGIN is retrieved (epsin is the database expression that *e* assigns to the first EPIC pointer file opened). The value of this attribute is stored for use in the PPLUS global symbol named eps$atrr. A list of the EPIC system attributes can be found in the EPIC system documentation.

In the next example, PPLUS e commands are used to create individual 1-d epic-formatted time series from a 2-d file containing moored Acoustic Doppler current measurements. The input file contains water velocity data at a single latitude,longitude as a function of depth and time. Output 1-d data files are time series, one for each of the selected depths. This example also makes use of PPLUS symbol substitution (note the PPLUS symbol zindx), and the special function ($integer), which are described in the PPLUS manual (see the chapter on PPLUS Command Files).

```
ppl>e openr "rd_600.vel_cdf"
ppl>e nextr
ppl>c
ppl>c Open the output pointer file for files in Classic EPIC
                      format
ppl>c
ppl>e openw "series.dat" "EPIC"
ppl>e nextr
ppl>c
ppl>c Select out the 5th through the 8th depth levels (and all
                      times)
ppl>c Write these out as individual time series for each
                      depth.
```

```
ppl>c
ppl>set zndx 5
ppl>while zndx .le. 8 then
ppl>c
ppl> e u1=eps1205[t=*,k='zndx']
ppl> e v1=eps1206[t=*,k='zndx']
ppl> e u1.varid=1205
ppl> e v1.varid=1206
ppl> e nextw "uv.'zndx'"
ppl> e write u1
ppl> e write v1
ppl> set zndx 'zndx' + 2.
ppl> set zndx $integer(zndx)
ppl>c
ppl>endw
ppl>e close epsin
ppl> close epsout
```

In the next example, a field of data is written out to an EPIC data file. The two components of wind are read in, and then wind speed is calculated. Since the output variable, wind speed, is not the same as the input variables, it is assigned an attribute VARID, which is the EPIC variable key code. In this example, wind speed is calculated, and then written out in an EPIC formatted file. The EPIC variable code for wind speed in meters/sec is 401. For a complete list of EPIC variable codes, see the EPIC system documentation.

```
ppl>e openr "t95w.dat"
ppl>e nextr
ppl>e openw "out_speed.dat" "EPIC"
ppl>e nextw "out_file.001"
ppl>e u1=u[t=*]
ppl>e v1=v[t=*]
ppl>e wspeed=sqrt(u1^2+v1^2)
ppl>e wspeed.VARID=401
ppl>e write wspeed
ppl>e close epsout
```

In the following example, data is read from a 3-d EPIC netCDF file, containing sub-surface water temperature data at several depths from several buoys deployed from 5N to 5S along 110W. An area-filled contour plot is made of the data at 2N. Note the use of the PPLUS global symbol named ppl$ylen (see the PPLUS manual chapter on PPLUS Command Files). Note use of the e command "status" to set PPLUS global symbols named eps$fld_tmin and eps$fld_tmax, and their use in the PPLUS time command to set the time axis extrema. Also note the use of the e command named whatis to see pertinent information about the field named t1.

```
ppl>e openr "/users/dai/data/t110w.cdf"
ppl>e nextr
ppl> t1=temp[t=*,z=*,y=2N]
ppl>c
```

```
ppl>c Use “whatis” to see pertinent information about variable
                      t1
ppl>c
ppl>e whatis t1
ppl>e t1
ppl>c
ppl>c Set up time axis
ppl>c
ppl>e status t1
ppl>taxis,on
ppl>time,w’eps$fld_tmin’,w’eps$fld_tmax’,w’eps$fld_tmin’
ppl>c
ppl>c Color bar and contour levels
ppl>c
ppl>set xp ‘ppl$xlen’ + .25
ppl>cbaxis,10,35,5
ppl>cblint,1
ppl>colorbar:nouser ‘xp’ 0 1 ‘ppl$ylen’
ppl>c
ppl>lev () (10,27,1,-3)(27,35,1,-3)
ppl>lev dark(10,25,5) dark(25,35,2)
ppl>lev (10,25,5,-1) (25,35,2,-1)
ppl>area
```

APPENDIX C

PPLUS Interface

e variables and PPLUS plot buffers

In the preceding example, data was never loaded into the PPLUS plotting data buffers, nor is it plotted. Assignment statements like temp=eps20[z=*] never load data into PPLUS. They just load data into the field variable name "temp." Assignment statements don't load data into PPLUS plot buffers. Other statements do load data into the PPLUS plot buffers. The following examples illustrate this.

```
temp=eps20[z=*]        is an assignment statement (does not
                       load into PPLUS plotting buffers)
(temp=eps20[z=*])      is not an assignment statement. It is a
                       field expression because it is enclosed
                       in parentheses (does load into PPLUS
                       plotting buffers)
temp*20                is not an assignment statement It is a
                       field expression (multiplies the field
                       temp by 20 and loads result into PPLUS
                       plotting buffers)
```

Loading PPLUS Buffers

PPLUS Symbols

Using the Calculator with PPLUS

APPENDIX D

Proposed New Features

netCDF Calculator API

The application programmers interface (API) for the netCDF calculator is designed to allow the calculator to be linked with and controlled by a user's application. Routines that initialize the calculator and send strings to be parsed by **nccalc** are provided. In addition, two routines, which are defined below, must be provided by the user.

nccalc routines

Four routines are provided to initialize **nccalc**, provide **nccalc** with a string to parse, provide **nccalc** data, and utility routines to free storage associated with **nccalc** structures. Additionally, four routines are provided to assist the user in getting information from **nccalc**. A header file ncc.h contains the necessary definitions for the following.

nccalc Control Routines

The **nccalc** control routines, along with the memory management routines are the commonly used commands provided by **nccalc**. They allow the user to initialize the queues and stacks, make **nccalc** commands, and check for errors.

void `ncc_init()`

ncc_init() is used to setup built in constants and functions to initialize command queue and data stack. The user's application must call ncc_init before using any other **nccalc** calls.

NccResult* `ncc_command(`**NccMessage*** `lines)`

`lines` structure containing one or more strings to be parsed by **nccalc.**

ncc_command() is the way that commands are sent to **nccalc**. The user's application creates an NccMessage which contains the command for **nccalc** to carry out. Normal commands will be only one line, but this structure leaves the option for multiple line commands. **nccalc** takes this line sent to it, parses it and performs the function. If during the parsing additional input is required **nccalc** will request additional input via the ncc_nexline() or ncc_request routines. After completing the command, **nccalc** will return a pointer to a structure of type **NccResult** explained below. The strings must be NULL terminated.

char* `ncc_error_message`

External character string that points to an error message if the NccResult returned from an ncc_command() call has the error flag set.

Memory Management

The data sent back to the user's application from the ncc_command becomes the responsibility of the application. This includes the responsibility for freeing up the space used by it. These commands assist the user in freeing some of the more complicated structures.

void `ncc_freeresult(`**NccResult*** `result)`

ncc_freeresult() is a routine provided to aid in reclaiming the storage associated with results passed to the user from the ncc_command routine. This routine

will free the memory used by the result structure as well as any structure it points to.

void ncc_freefield(**NccField*** field)

ncc_freefield() should be used to free a field sent back in a NccResult structure that no longer has a result pointing to it. It will free the memory used by the field as well as all other arrays and structures it points to.

void ncc_freemsg(**NccMessage*** msg)

ncc_freemsg() is a routine which aids in freeing the space used by the NccMessage type. This routine follows the linked list and frees all of the memory used by it. It is not necessary to use this routine if ncc_freeresult has been run on a structure which points to the message.

Load User Field

The user does not always want to load information the way **nccalc** does it and from the sources **nccalc** handles. This routine, therefore allows the user a method for putting data into **nccalc** that it created or received in some other way.

void ncc_put_field(**char*** name, **NccField*** fld)

Allows the user's application to directly load a field into **nccalc.**

Get ncc Data

Although ncc_command returns the data it creates in it's NccResult construct, there may be times that the user application wants to query **nccalc** to get the data that it is storing. These routines allow the application to directly query the value of a variable that has been set in **nccalc**.

double ncc_get_scalar(**char*** name)

This routine returns the value of the scalar variable with the given name.

NccField* ncc_get_field(**char*** name)

Returns the value of the field with the given name.

char* `ncc_get_string`(**char*** `name`)

Returns the value of the string with the given name.

NccVector* `ncc_get_vector`(**char*** `name`)

Returns the value of the vector with the given name.

User routines

Two routines must be written by the user to provide **nccalc** with additional input. The ncc.h header file contains the definition of the **nccalc** structures and function prototypes.

NccMessage* `ncc_nextline()`

If **nccalc** requires additional input in order to complete a syntactical construction ncc_nextline() is called by **nccalc**. ncc_nextline() allows the user to input multiple line procedure definitions, if-then-else constructs, etc. Each string must be null terminated, and the last line must have a NULL next pointer.

char* `ncc_request(`**NccMessage*** `prompt)`

promptStructure containging strings used to prompt the user for input.

ncc_request() returns a null terminated string to **nccalc** containing the user response to the prompt string. **nccalc** will convert the string to double, long, or character as necessary.

Data Structures

NccResult

The ncc_command() function returns to the user's application an NccResult structure. This structure is shown below

```
typedef struct NccResult {
```

```
  int type;           /* type of data returned */
  int error;          /* error indication */
  union {
    double val;       /* scalar value */
    NccField *fld;    /* pointer to field */
    NccVector *vect;  /* pointer to vector */
    char *str;        /* pointer to character string */
    NccMessage *msg;  /* pointer to message */
    } u;
  NccResult *next;    /* pointer to next result */
} NccResult;
```

The value returned in type is equal to one of the following: NCC_MESSAGE, NCC_VECTOR, NCC_FIELD, NCC_SCALAR, NCC_STRING, or NCC_NONE. By looking at this type, the user's application can tell which part of the union is valid.

NCC_MESSAGE is used to pass the user character messages, such as in response to a *whatis* command.

NCC_NONE is returned when the command does not return anything, such as a slab assignment.

The error element returns a non-zero result if there were any problems.

NccField

```
typedef struct NccField {
  char* name;         /* name of field */
  char* long_name;    /* long name of field */
  char* units;        /* units of field */
  int dims[4];        /* dimensions of storage */
  float* arr;         /* pointer to field storage */
  float* axis[4];     /* pointer to axis storage */
  long* taxis;        /* pointer to t-axis storage */
  long t_origin[2];   /* origin of t-axis in eptime format */
  char* axname[4];    /* names of axes */
  char* axunits[4];   /* units of axes */
} NccField;
```

If axis[3] is a NULL pointer then taxis contains valid information, otherwise, axis[3] contains the axis values.

NccVector

```
typedef struct NccVector {
  NccField* xcomp;    /* the vector's x component */
```

```
  NccField* ycomp;    /* the vector's y component */
  NccField* zcomp;    /* the vector's z component */
} NccVector;
```

If any component is missing, the corresponding vector component will be NULL

NccMessage

```
typedef struct NccMessage {
  char* str;          /* one line of the message */
  NccMessage* next;   /* a pointer to the next line */
} NccMessage;
```

Each line of the message (*str*) must be NULL terminated. The *next* pointer in the last line of the message is NULL.

Inquiry commands

Additional whatis commands look like:

| ***whatis*** *whopt field-name*
| ***whatis*** *whopt fld-var*
| ***whatis*** *whopt db-var*
| ***whatis*** *whopt slab-var*
| ***whatis*** *whopt fld-var* **.** *attr_name*
| ***whatis*** *whopt db-var* **.** *attr_name*
| ***whatis*** *whopt* **.** *attr_name*
| ***whatis*** *whopt vect-var*
| ***whatis*** *whopt var*

TABLE 19. whatis actions.

variable name	action
field-name	returns a description of the field
field-var	returns a description of the field pointed to by field-var
db-var	returns information about the database pointed to by db-var
slab-var	returns the present value of the slab dimensions
*field-var***.***attr_name*	returns information about the given attribute
*db-var***.***attr_name*	returns information about the attribute given
.*attr_name*	returns information about the given attribute

TABLE 19. whatis actions.

variable name	action
vect-var	returns information about the given vector
var	returns the value of the variable

The whatis function returns information about the object given. The kind of information is dependent on the type of element it is. The amount of information given is dependent upon the options listed below

TABLE 20. *whopt* values.

whopt	action
-f	full - displays all available information about the object
-b	brief - displays very little about object

Note that if your element in question is a *var*, -f and -b have no effect.

More Field Functions

TABLE 21. More field functions.

function	description
regression()	

regression returns?

One more scalar function.

TABLE 22. Modulo function

expr1 % expr2	returns expr1 modulo expr2

Vector expressions

Triplets of *field-name* or *fld-var* can be logically bound into a single *vect-var.* The vector creation syntax is:

vector *vect-var* ***(*** *fld-var* ***,*** *fld-var* ***,*** *fld-var* ***)***

vector *vect-var* ***(*** *field-name* ***,*** *field-name* ***,*** *field-name* ***)***

The components of vectors are not required to be on the same grid. However, several of the vector functions do have this requirement. Because vectors are *logical* constructs a vector must be explicitly created before values can be assigned to it. For example,

```
field gx              Declare field gx
field gy              Declare field gy
field gz              Declare field gz
vector tgrad(gx, gy, gz)Bind gx,gy,gz into vector tgrad
tgrad = grad(t)       Compute the gradient of t.
```

vexpr: *vect-var*
| *vect-var* ***[*** *slexpr* ***]***
| ***(*** *vexpr* ***)***
| *vexpr binop vexpr*
| *vexpr binop expr*
| *expr binop vexpr*
| *function* ***(*** *fexpr* ***)***
| *function* ***(*** *vexpr* ***)***
| *function* ***(*** *vexpr* ***,*** *vexpr* ***)***

binop refers to binary operators such as addition and are done vector component by vector component and field element by field element. Vector components do not have to be on the same grid for the vector binary operators.

TABLE 23. Vector operators in decreasing order of precedence.

binop	description
^	exponentiation (FORTRAN **), right associative
/ *	division, multiplication
+ -	addition, subtraction

Exponentiation is only valid when raising a vector to a scalar power. The other *binop*'s are valid for any combination of fields and scalars.

TABLE 24. Built-in vector functions.

function	description
curl(v)	$\nabla\times$, curl of vector v
grad(f)	∇ , gradient of field f
cross(v1,v2)	$v1 \times v2$, vector cross product of vectors v1 with v2
absv(v)	| v |, absolute value of vector v
expv(v)	e^v, exponential of vector v
logv(v)	ln v, logarithm base *e* of vector v
log10v(v)	log v, logarithm base 10 of vector v
sqrtv(v)	$\sqrt{v}$,square root of vector v
avev(v,range)	compute vector averaged over range
primev(v,range)	compute perturbation vector by averaging over range

`curl` and `cross` require the vector components to be on the same grid.

TABLE 25. Built-in field functions with vector arguments.

function	description
div(v)	$\nabla\bullet$, divergence of vector v
dot(v1,v2)	$v1 \bullet v2$, dot product vectors v1 and v2.

Vector expressions

Triplets of *field-name* or *fld-var* can be logically bound into a single *vect-var.* The vector creation syntax is:

vector *vect-var* **(** *fld-var* **,** *fld-var* **,** *fld-var* **)**

vector *vect-var* **(** *field-name* **,** *field-name* **,** *field-name* **)**

The components of vectors are not required to be on the same grid. However, several of the vector functions do have this requirement. Because vectors are *logical* constructs a vector must be explicitly created before values can be assigned to it. For example,

```
field gx              Declare field gx
```

```
field gy              Declare field gy
field gz              Declare field gz
vector tgrad(gx, gy, gz)Bind gx,gy,gz into vector tgrad
tgrad = grad(t)       Compute the gradient of t.
```

vexpr: *vect-var*
| *vect-var* **[** *slexpr* **]**
| **(** *vexpr* **)**
| *vexpr binop vexpr*
| *vexpr binop expr*
| *expr binop vexpr*
| *function* **(** *fexpr* **)**
| *function* **(** *vexpr* **)**
| *function* **(** *vexpr* **,** *vexpr* **)**

binop refers to binary operators such as addition and are done vector component by vector component and field element by field element. Vector components do not have to be on the same grid for the vector binary operators.

TABLE 26. Vector operators in decreasing order of precedence.

binop	description
^	exponentiation (FORTRAN **), right associative
/ *	division, multiplication
+ -	addition, subtraction

Exponentiation is only valid when raising a vector to a scalar power. The other *binop*'s are valid for any combination of fields and scalars.

TABLE 27. Built-in vector functions.

function	description
curl(v)	$\nabla\times$, curl of vector v
grad(f)	∇ , gradient of field f
cross(v1,v2)	$v1 \times v2$, vector cross product of vectors v1 with v2
abs(v)	| v |, absolute value of vector v
exp(v)	e^v, exponential of vector v
log(v)	ln v, logarithm base *e* of vector v
log10(v)	log v, logarithm base 10 of vector v
sqrt(v)	$\sqrt{v}$,square root of vector v

TABLE 27. Built-in vector functions.

function	description
ave(v,range)	compute vector averaged over range
prime(v,range)	compute perturbation vector by averaging over range

`curl` and `cross` require the vector components to be on the same grid.

TABLE 28. Built-in field functions with vector arguments.

function	description
div(v)	$\nabla\bullet$, divergence of vector v
dot(v1,v2)	$v1 \bullet v2$, dot product vectors v1 and v2.

Grid generation

The creation of grids that do not already exist as a netCDF variable can be useful in accurately reproducing the finite difference expression used in a model. Grids can be either regular (evenly spaced points in a dimension) or irregular (unevenly spaced points in a dimension) or mixed. Axis expression grammar is:

axexpr: *ax-var*
| **axis (** *fexpr* **, [** *slexpr* **])**
| **axis (** *expr* **:** *expr* **:** *expr* **)**
| **axis (** *axlist* **)**

The syntax for an axis list is:

axlist: *number*
| *axlist* **,** *number*

The grammar for field creation is:

field *fld-var* **with** *axexpr*
field *fld-var* **like** *fld-var*

The additional stmt type will be added

stmt: *fld-var [slexpr] = fexpr*

I/O - interaction with the keyboard

There is a requirement to be able to interact with the user, to have the calculator ask questions of the user when there is not enough information provided to fulfill the task. This will be done differently for the different interfacing programs using the calculator. Every change here will contain an ifdef loop dependant on whether the system is running through PPLUS, stand-alone, or with another program. We are also toying with using remote procedure calls to do this communication.

Conditional evaluation

The where grammar is:

where (*fexpr* ***)*** *stmt*
| ***where (*** *fexpr* ***)*** *stmt* ***else*** *stmt*

The where statement is similar to the if statement, except that it does a test on every element of the field rather than the field itself. For example

```
where (a<0) a=0 else a=1
```

goes through field a, and changes every element in it to 0 or 1 depending on whether it is <0 or not. As with if, it is also valid to use multiple lines:

```
where (a<0) {
    a=0
} else {
    a=1
}
```

Index